

高级运算符 (Advanced Operators)

1.0 翻译: xielingwang (https://github.com/xielingwang) 校对: numbbbbb (https://github.com/numbbbbb)

2.0 翻译+校对: buginux (https://github.com/buginux)

2.1 校对: shanks (http://codebuild.me), 2015-11-01

2.2 翻译+校对: SketchK (https://github.com/SketchK) 2016-05-17

本页内容包括：

- 位运算符
- 溢出运算符
- 优先级和结合性
- 运算符函数
- 自定义运算符

除了在之前介绍过的基本运算符 (./02_Basic_Operators.html)，Swift 中还有许多可以对数值进行复杂运算的高级运算符。这些高级运算符包含了在 C 和 Objective-C 中已经被大家所熟知的位运算符和移位运算符。

与 C 语言中的算术运算符不同，Swift 中的算术运算符默认是不会溢出的。所有溢出行为都会被捕获并报告为错误。如果想让系统允许溢出行为，可以选择使用 Swift 中另一套默认支持溢出的运算符，比如溢出加法运算符（&+）。所有的这些溢出运算符都是以 & 开头的。

自定义结构体、类和枚举时，如果也为它们提供标准 Swift 运算符的实现，将会非常有用。在 Swift 中自定义运算符非常简单，运算符也会针对不同类型使用对应实现。

我们不用被预定义的运算符所限制。在 Swift 中可以自由地定义中缀、前缀、后缀和赋值运算符，以及相应的优先级与结合性。这些运算符在代码中可以像预定义的运算符一样使用，我们甚至可以扩展已有的类型以支持自定义的运算符。

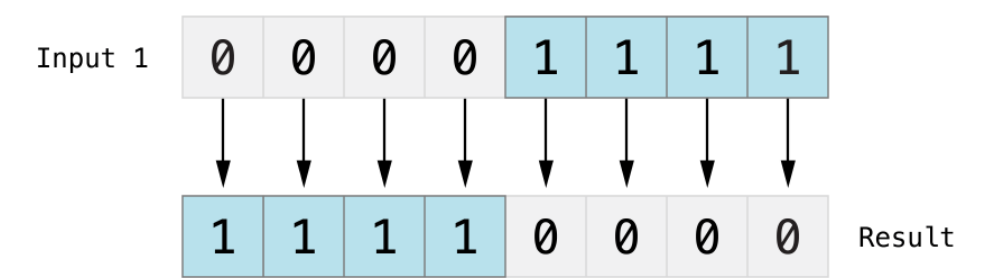
位运算符

位运算符可以操作数据结构中每个独立的比特位。它们通常被用在底层开发中，比如图形编程和创建设备驱动。位运算符在处理外部资源的原始数据时也十分有用，比如对自定义通信协议传输的数据进行编码和解码。

Swift 支持 C 语言中的全部位运算符，接下来会——介绍。

按位取反运算符

按位取反运算符（~）可以对一个数值的全部比特位进行取反：



按位取反运算符是一个前缀运算符，需要直接放在运算的数之前，并且它们之间不能添加任何空格：

```
let initialBits: UInt8 = 0b0001111
let invertedBits = ~initialBits // 等于 0b11110000
```

UInt8 类型的整数有 8 个比特位，可以存储 0 ~ 255 之间的任意整数。这个例子初始化了一个 UInt8 类型的整数，并赋值为二进制的 0001111，它的前 4 位都为 0，后 4 位都为 1。这个值等价于十进制的 15。

接着使用按位取反运算符创建了一个名为 invertedBits 的常量，这个常量的值与全部位取反后的 initialBits 相等。即所有的 0 都变成了 1，同时所有的 1 都变成 0。invertedBits 的二进制值为 11110000，等价于无符号十进制数的 240。

按位与运算符

按位左移运算符 (<<) 和按位右移运算符 (>>) 可以对一个数的所有位进行指定位数的左移和右移，但是需要遵守下面定义的规则。

> iOS > The Swift Programming Language 中文版

对于一个数进行按位左移或按位右移，相当于对这个数进行乘以 2 或除以 2 的运算。将一个整数左移一位，等价于将这个数乘以 2，同样地，将一个整数右移一位，等价于将这个数除以 2。

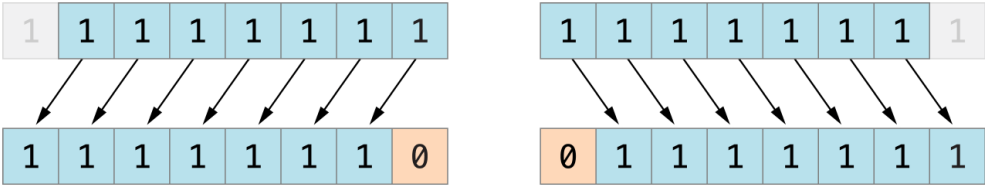
无符号整数的移位运算

对无符号整数进行移位的规则如下：

- 1. 已经存在的位按指定的位数进行左移和右移。
- 2. 任何因移动而超出整型存储范围的位都会被丢弃。
- 3. 用 0 来填充移位后产生的空白位。

这种方法称为逻辑移位。

以下这张图展示了 11111111 << 1（即把 11111111 向左移动 1 位），和 11111111 >> 1（即把 11111111 向右移动 1 位）的结果。蓝色的部分是被移位的，灰色的部分是被抛弃的，橙色的部分则则是被填充进来的：



下面的代码演示了 Swift 中的移位运算：

```
let shiftBits: UInt8 = 4 // 即二进制的 00001000
shiftBits << 1           // 00001000
shiftBits << 2           // 00010000
shiftBits << 5           // 10000000
shiftBits << 6           // 00000000
shiftBits >> 2           // 00000001
```

可以使用移位运算对其他的数据类型进行编码和解码：

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16 // redComponent 是 0xCC，即 204
let greenComponent = (pink & 0x00FF00) >> 8 // greenComponent 是 0x66，即 102
let blueComponent = pink & 0x0000FF // blueComponent 是 0x99，即 153
```

这个示例使用了一个命名为 pink 的 UInt32 型常量来存储 CSS 中粉色的颜色值。该 CSS 的十六进制颜色值 #CC6699，在 Swift 中表示为 0xCC6699。然后利用按位与运算符 (&) 和按位右移运算符 (>>) 从这个颜色值中分解出红 (CC)、绿 (66) 以及蓝 (99) 三个部分。

红色部分是通过 0xCC6699 和 0xFF0000 进行按位与运算后得到的。0xFF0000 中的 0 部分“掩盖”了 0xCC6699 中的第二、第三个字节，使得数值中的 6699 被忽略，只留下 0xCC0000。

然后，再将这个数按向右移动 16 位 (>> 16)。十六进制中每两个字符表示 8 个比特位，所以移动 16 位后 0xCC0000 就变为 0x0000CC。这个数和 0xCC 是等同的，也就是十进制数值的 204。

同样的，绿色部分通过对 0xCC6699 和 0x00FF00 进行按位与运算得到 0x006600。然后将这个数向右移动 8 位，得到 0x66，也就是十进制数值的 102。

最后，蓝色部分通过对 0xCC6699 和 0x0000FF 进行按位与运算得到 0x000099。这里不需要再向右移位，所以结果为 0x99，也就是十进制数值的 153。

有符号整数的移位运算

对比无符号整数，有符号整数的移位运算相对复杂得多，这种复杂性源于有符号整数的二进制表现形式。（为了简单起见，以下的示例都是基于 8 比特位的有符号整数的，但是其中的原理对任何位数的有符号整数都是通用的。）

有符号整数使用第 1 个比特位（通常被称为符号位）来表示这个数的正负。符号位为 0 代表正数，为 1 代表负数。其余的比特位（通常被称为数值位）存储了实际的值。有符号正整数和无符号数的存储方式是一样的，都是从 0 开始算起。这是值为 4 的 Int8 型整数的二进制位表现形式：

符号位为 0，说明这是一个正数，另外 7 位则代表了十进制数值 4 的二进制表示。

负数的存储方式略有不同。它存储的值的绝对值等于 2 的 n 次方减去它的实际值（也就是数值位表示的值），这里的 n 为数值位的比特位数。一个 8 比特位的数有 7 个比特位是数值位，所以是 2 的 7 次方，即 128。

这是值为 -4 的 Int8 型整数的二进制位表现形式：

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

负数的表示通常被称为二进制补码表示。用这种方法来表示负数乍看起来有点奇怪，但它有几个优点。

首先，如果想对 -1 和 -4 进行加法运算，我们只需要将这两个数的全部 8 个比特位进行相加，并且将计算结果中超出 8 位的数值丢弃：

其次，使用二进制补码可以使负数的按位左移和右移运算得到跟正数同样的效果，即每向左移一位就将自身的数值乘以 2，每向右一位就将自身的数值除以 2。要达到此目的，对有符号整数的右移有一个额外的规则：

- 当对整数进行按位右移运算时，遵循与无符号整数相同的规则，但是对于移位产生的空白位使用符号位进行填充，而不是用 0。

这个行为可以确保有符号整数的符号位不会因为右移运算而改变，这通常被称为算术移位。

由于正数和负数的特殊存储方式，在对它们进行右移的时候，会使它们越来越接近 0。在移位的过程中保持符号位不变，意味着负整数在接近 0 的过程中会一直保持为负。

溢出运算符

在默认情况下，当向一个整数赋予超过它容量的值时，Swift 默认会报错，而不是生成一个无效的数。这个行为为我们在运算过大或着过小的数的时候提供了额外的安全性。

例如，Int16 型整数能容纳的有符号整数范围是 -32768 到 32767，当为一个 Int16 型变量赋的值超过这个范围时，系统就会报错：

```
var potentialOverflow = Int16.max
// potentialOverflow 的值是 32767，这是 Int16 能容纳的最大整数
potentialOverflow += 1
// 这里会报错
```

为过大或者过小的数值提供错误处理，能让我们在处理边界值时更加灵活。

然而，也可以选择让系统在数值溢出的时候采取截断处理，而非报错。可以使用 Swift 提供的三个溢出运算符来让系统支持整数溢出运算。这些运算符都是以 & 开头的：

- 溢出加法 &+
- 溢出减法 &-
- 溢出乘法 &*

数值溢出

数值有可能出现上溢或者下溢。

这个示例演示了当我们对一个无符号整数使用溢出加法（&+）进行上溢运算时会发生什么：

```
var unsignedOverflow = UInt8.max
// unsignedOverflow 等于 UInt8 所能容纳的最大整数 255
unsignedOverflow = unsignedOverflow &+ 1
// 此时 unsignedOverflow 等于 0
```

unsignedOverflow 被初始化为 UInt8 所能容纳的最大整数（255，以二进制表示即 11111111）。然后使用了溢出加法运算符（&+）对其进行加 1 运算。这使得它的二进制表示正好超出 UInt8 所能容纳的位数，也就导致了数值的溢出，如下图所示。数值溢出后，留在 UInt8 边界内的值是 00000000，也就是十进制数值的 0。

同样地，当我们对一个无符号整数使用溢出减法（&-）进行下溢运算时也会产生类似的现象：

```
var unsignedOverflow = UInt8.min
// unsignedOverflow 等于 UInt8 所能容纳的最小整数 0
unsignedOverflow = unsignedOverflow &- 1
// 此时 unsignedOverflow 等于 255
```

UInt8 型整数能容纳的最小值是 0，以二进制表示即 00000000。当使用溢出减法运算符对其进行减 1 运算时，数值会产生下溢并被截断为 11111111，也就是十进制数值的 255。

```
var signedOverflow = Int8.min
// signedOverflow 等于 Int8 所能容纳的最小整数 -128
signedOverflow = signedOverflow &- 1
// 此时 signedOverflow 等于 127
```

Int8 型整数能容纳的最小值是 -128，以二进制表示即 10000000。当使用溢出减法运算符对其进行减 1 运算时，符号位被翻转，得到二进制数值 01111111，也就是十进制数值的 127，这个值也是 Int8 型整数所能容纳的最大值。

对于无符号与有符号整型数值来说，当出现上溢时，它们会从数值所能容纳的最大数变成最小的数。同样地，当发生下溢时，它们会从所能容纳的最小数变成最大的数。

优先级和结合性

运算符的优先级使得一些运算符优先于其他运算符，高优先级的运算符会先被计算。

结合性定义了相同优先级的运算符是如何结合的，也就是说，是与左边结合为一组，还是与右边结合为一组。可以将这意思理解为“它们是与左边的表达式结合的”或者“它们是与右边的表达式结合的”。

在复合表达式的运算顺序中，运算符的优先级和结合性是非常重要的。举例来说，运算符优先级解释了为什么下面这个表达式的运算结果会是 17。

```
2 + 3 % 4 * 5
// 结果是 17
```

如果完全从左到右进行运算，则运算的过程是这样的：

- 2 + 3 = 5
- 5 % 4 = 1
- 1 * 5 = 5

但是正确答案是 17 而不是 5。优先级高的运算符要先于优先级低的运算符进行计算。与 C 语言类似，在 Swift 中，乘法运算符（*）与取余运算符（%）的优先级高于加法运算符（+）。因此，它们的计算顺序要先于加法运算。

而乘法与取余的优先级相同。这时为了得到正确的运算顺序，还需要考虑结合性。乘法与取余运算都是左结合的。可以将这考虑成为这两部分表达式都隐式地加上了括号：

```
2 + ((3 % 4) * 5)
```

(3 % 4) 等于 3，所以表达式相当于：

```
2 + (3 * 5)
```

3 * 5 等于 15，所以表达式相当于：

```
2 + 15
```

因此计算结果为 17。

如果想查看完整的 Swift 运算符优先级和结合性规则，请参考表达式 (../chapter3/04_Expressions.html)。如果想查看 Swift 标准库提供所有的运算符，请查看 Swift Standard Library Operators Reference (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_StandardLibrary_Operators

注意

相对 C 语言和 Objective-C 来说，Swift 的运算符优先级和结合性规则更加简洁和可预测。但是，这也意味着它们相较于 C 语言及其衍生语言并不是完全一致的。在对现有的代码进行移植的时候，要注意确保运算符的行为仍然符合你的预期。

运算符函数

类和结构体可以为现有的运算符提供自定义的实现，这通常被称为运算符重载。

下面的例子展示了如何为自定义的结构体实现加法运算符（+）。算术加法运算符是一个双目运算符，因为它可以对两个值进行运算，同时它还是中缀运算符，因为它出现在两个值中间。

例子中定义了一个名为 Vector2D 的结构体用来表示二维坐标向量（x，y），紧接着定义了一个可以对两个 Vector2D 结构体进行相加的运算符函数：

2016/7/12		高级运算符（Advanced Operators） - The Swift Programming Language 中文版 - 极客学院Wiki	
<div><div>极客学院</div><div>jikexueyuan.com</div><div>(http://www.jikexueyuan.com)</div></div> <div>关于 (http://wiki.jikexueyuan.com/project/swift/)</div> <div>欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)</div> <div>Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)</div> <div>基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)</div> <div>基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)</div> <div>字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)</div> <div>集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)</div> <div>Via 由 [极客学院 Wiki</div> <div>(http://wiki.jikexueyuan.com)]</div> <div>提供</div>		Wiki >	<div><div>移动开发 iOS, Android, Swift</div><div>The Swift Programming Language 中文版</div></div> <div><pre>struct Vector2D { let x: Double, y: Double } func + (left: Vector2D, right: Vector2D) -> Vector2D { return Vector2D(x: left.x + right.x, y: left.y + right.y) }</pre></div> <div>该运算符函数被定义为一个全局函数，并且函数的名字与它要进行重载的 + 名字一致。因为算术加法运算符是双目运算符，所以这个运算符函数接收两个类型为 Vector2D 的参数，同时有一个 Vector2D 类型的返回值。</div> <div>在这个实现中，输入参数分别被命名为 left 和 right，代表在 + 运算符左边和右边的两个 Vector2D 实例。函数返回了一个新的 Vector2D 实例，这个实例的 x 和 y 分别等于作为参数的两个实例的 x 和 y 的值之和。</div> <div>这个函数被定义成全局的，而不是 Vector2D 结构体的成员方法，所以任意两个 Vector2D 实例都可以使用这个中缀运算符：</div> <div><pre>let vector = Vector2D(x: 3.0, y: 1.0) let anotherVector = Vector2D(x: 2.0, y: 4.0) let combinedVector = vector + anotherVector // combinedVector 是一个新的 Vector2D 实例，值为 (5.0, 5.0)</pre></div> <div>这个例子实现两个向量 (3.0, 1.0) 和 (2.0, 4.0) 的相加，并得到新的向量 (5.0, 5.0)。这个过程如下图所示：</div>
		<h3>前缀和后缀运算符</h3> <p>上个例子演示了一个双目中缀运算符的自定义实现。类与结构体也能提供标准单目运算符的实现。单目运算符只运算一个值。当运算符出现在值之前时，它就是前缀的（例如 -a），而当它出现在值之后时，它就是后缀的（例如 b!）。</p> <p>要实现前缀或者后缀运算符，需要在声明运算符函数的时候在 func 关键字之前指定 prefix 或者 postfix 修饰符：</p>	
		<pre>prefix func - (vector: Vector2D) -> Vector2D { return Vector2D(x: -vector.x, y: -vector.y) }</pre>	
		<p>这段代码为 Vector2D 类型实现了单目负号运算符。由于该运算符是前缀运算符，所以这个函数需要加上 prefix 修饰符。</p> <p>对于简单数值，单目负号运算符可以对它们的正负性进行改变。对于 Vector2D 来说，该运算将其 x 和 y 属性的正负性都进行了改变：</p>	
		<pre>let positive = Vector2D(x: 3.0, y: 4.0) let negative = -positive // negative 是一个值为 (-3.0, -4.0) 的 Vector2D 实例 let alsoPositive = -negative // alsoPositive 是一个值为 (3.0, 4.0) 的 Vector2D 实例</pre>	
		<h3>复合赋值运算符</h3> <p>复合赋值运算符将赋值运算符 (=) 与其它运算符进行结合。例如，将加法与赋值结合成加法赋值运算符 (+=)。在实现的时候，需要把运算符的左参数设置成 inout 类型，因为这个参数的值会在运算符函数内直接被修改。</p>	
		<pre>func += (inout left: Vector2D, right: Vector2D) { left = left + right }</pre>	
		<p>因为加法运算在之前已经定义过了，所以在这里无需重新定义。在这里可以直接利用现有的加法运算符函数，用它来对左值和右值进行相加，并再次赋值给左值：</p>	
		<pre>var original = Vector2D(x: 1.0, y: 2.0) let vectorToAdd = Vector2D(x: 3.0, y: 4.0) original += vectorToAdd // original 的值现在为 (4.0, 6.0)</pre>	
		<div>注意</div> <div>不能对默认的赋值运算符 (=) 进行重载。只有组合赋值运算符可以被重载。同样地，也无法对三目条件运算符 (a ? b : c) 进行重载。</div>	

等价运算符

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

自定义的类和结构体没有对等价运算符进行默认实现，等价运算符通常被称为“相等”运算符（`==`）与“不等”运算符（`!=`）。对于自定义类型，Swift 无法判断其是否“相等”，因为“相等”的含义取决于这些自定义类型在你的代码中所扮演的角色。

为了使用等价运算符能对自定义的类型进行判等运算，需要为其提供自定义实现，实现的方法与其它中缀运算符一样：

```
func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}
func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
```

上述代码实现了“相等”运算符（`==`）来判断两个 `Vector2D` 实例是否相等。对于 `Vector2D` 类型来说，“相等”意味着“两个实例的 `x` 属性和 `y` 属性都相等”，这也是代码中用来进行判等的逻辑。示例里同时也实现了“不等”运算符（`!=`），它简单地将“相等”运算符的结果进行取反后返回。

现在我们可以使用这两个运算符来判断两个 `Vector2D` 实例是否相等：

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// 打印 “These two vectors are equivalent.”
```

自定义运算符

除了实现标准运算符，在 Swift 中还可以声明和实现自定义运算符。可以用来自定义运算符的字符列表请参考运算符（`./chapter3/02_Lexical_Structure.html#operators`）。

新的运算符要使用 `operator` 关键字在全局作用域内进行定义，同时还要指定 `prefix`、`infix` 或者 `postfix` 修饰符：

```
prefix operator +++ {}
```

上面的代码定义了一个新的名为 `+++` 的前缀运算符。对于这个运算符，在 Swift 中并没有意义，因此我们针对 `Vector2D` 的实例来定义它的意义。对这个示例来讲，`+++` 被实现为“前缀双自增”运算符。它使用了前面定义的复合加法运算符来让矩阵对自身进行相加，从而让 `Vector2D` 实例的 `x` 属性和 `y` 属性的值翻倍：

```
prefix func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

`Vector2D` 的 `+++` 的实现和 `++` 的实现很相似，唯一不同的是前者对自身进行相加，而后者是与另一个值为 `(1.0, 1.0)` 的向量相加。

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled 现在的值为 (2.0, 8.0)
// afterDoubling 现在的值也为 (2.0, 8.0)
```

自定义中缀运算符的优先级和结合性

自定义的中缀运算符也可以指定优先级和结合性。优先级和结合性中详细阐述了这两个特性是如何对中缀运算符的运算产生影响的。

结合性可取的值得 `left`，`right` 和 `none`。当左结合运算符跟其他相同优先级的左结合运算符写在一起时，会跟左边的值进行结合。同理，当右结合运算符跟其他相同优先级的右结合运算符写在一起时，会跟右边的值进行结合。而非结合运算符不能跟其他相同优先级的运算符写在一起。

结合性的默认值是 `none`，优先级的默认值 `100`。

以下例子定义了一个新的中缀运算符 `++`，此运算符的结合性为 `left`，并且它的优先级为 `140`：

2016/7/12

极客学院

jikexueyuan.com

(http://www.jikexueyuan.com)

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki (http://wiki.jikexueyuan.com)] 提供

高级运算符 (Advanced Operators) - The Swift Programming Language 中文版 - 极客学院Wiki

移动开发 (iOS > The Swift Programming Language 中文版)

```
infix operator + - { associativity left precedence 140 }
func +(left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector + - secondVector
// plusMinusVector 是一个 Vector2D 实例, 并且它的值为 (4.0, -2.0)
```

这个运算符把两个向量的 x 值相加, 同时用第一个向量的 y 值减去第二个向量的 y 值。因为它本质上是属于“相加型”运算符, 所以将它的结合性和优先级被分别设置为 left 和 140, 这与 + 和 - 等默认的中缀“相加型”运算符是相同的。关于 Swift 标准库提供的运算符的结合性与优先级, 请参考 Swift Standard Library Operators Reference (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_StandardLibrary_Operators)

注意

当定义前缀与后缀运算符的时候, 我们并没有指定优先级。然而, 如果对同一个值同时使用前缀与后缀运算符, 则后缀运算符会先参与运算。

上一篇: 访问控制 (/project/swift/chapter2/24_Access_Control.html)

下一篇: 关于语言参考 (/project/swift/chapter3/01_About_the_Language_Reference.html)

被顶起来的评论



大明小书童 (http://t.qq.com/kjkqjll)

去年8月几号开始看的, 直到今天才看完, 历时半年。开始几章看得比较快。后来就有点疲了。断了一段时间, 然后强迫自己, 继续坚持看, 类oc的时候比较轻松, 而自身特性的时候, 虽然理解, 但不明白为何如此设计, 还需要大量的运用实践。这个语言表面上看似简洁, 大方, 其实独有特性之多, 超过C++、Java。几乎杂糅了主流脚本语言的特性。博采众长, 灵活运用起来就够复杂。虽说看一遍Swift初见也能码代码, 做项目。但是, swift之精髓, 还需假以时日。如果不能吃透, 可能还是用C++或者Java 的思想来做项目。那样就只是语法糖了。虽然, 译作中有些地方有点苦涩难懂, 然瑕不掩瑜, 仍是煌煌之作! 衷心感谢所有参与到这个翻译项目的辛勤付出者, 看过这本译作的人, 都会记住你们的努力! 人人为我, 我为人人, 以后有时间, 我也希望尽一份绵薄之力!

2月23日 回复 顶(1) 转发

37条评论 7条新浪微博

最新 最早 最热



Hummer

Mark一下

6月13日 回复 顶 转发



JeansH

回复 ningning: let num = Int8(bitPattern: 0b1000_0001)

6月11日 回复 顶 转发



ClavisJ

去年看到现在 终于看完了 感谢翻译

6月2日 回复 顶 转发



图片挂了 🙄

6月1日 回复 顶 转发



ningning

回复 ningning: 写错了, 最高位, 不是最好位

5月15日 回复 顶 转发



ningning

我不懂负数那一块什么意思, 负数怎么表示, 在playground 里面写 let num: Int8 = 0b10000001, 但报错, 说129超出了, 为什么是129? 最好位不是表示负数吗?

5月15日 回复 顶 转发



浪子潇润 (http://hujiaweibujidao.github.io/)

签到, 感谢! 🍊

(http://hujiaweibujidao.github.io/)5月9日 回复 顶 转发



那风吹过的回忆 (http://weibo.com/2189623234)

看完了, 签个到

(http://weibo.com/2189623234)5月4日 回复 顶 转发



小嘟嘟 (http://weibo.com/3015050631)

http://wiki.jikexueyuan.com/project/swift/chapter2/25_Advanced_Operators.html

8/9

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki

(http://wiki.jikexueyuan.com)]

提供

高级运算符（Advanced Operators） - The Swift Programming Language 中文版 - 极客学院Wiki



终于看完了，断断续续的，哎
明天开始看视频了

移动开发 > iOS > The Swift Programming Language 中文版
(http://weibo.com/3015050631)4月21日 回复 顶 转发



—Tyler (http://weibo.com/3234460720)
今天终于看完教程～好开心，非常感谢翻译人员的付出！

(http://weibo.com/3234460720)3月29日 回复 顶 转发



余檀池
谢谢所有翻译人员的付出，非常感谢

2月27日 回复 顶 转发



果核果核 (http://weibo.com/TechCorer)
打卡

(http://weibo.com/TechCorer)2月24日 回复 顶 转发



大明小书童 (http://t.qq.com/kjkqjll)
去年8月几号开始看的，直到今天才看完，历时半年。开始几章看得比较快。后来就有点累了。断了一段时间，然后强迫自己，继续坚持看，类oc的时候比较轻松，而自身特性的时候，虽然理解，但不明白为何如此设计，还需要大量的运用实践。这个语言表面上看似简洁，大方，其实独有特性之多，超过C++、Java。几乎杂糅了主流脚本语言的特性。博采众长，灵活运用起来就够复杂。虽说看一遍Swift初见也能码代码，做项目。但是，swift之精髓，还需假以时日。如果不能吃透，可能还是用C++或者Java 的思想来做项目。那样就只是语法糖了。虽然，译作中有些地方有点苦涩难懂，然瑕不掩瑜，仍是煌煌之作！衷心感谢所有参与到这个翻译项目的辛勤付出者，看过这本译作的人，都会记住你们的努力！人人为我，我为人人，以后有时间，我也希望尽一份绵薄之力！

(http://t.qq.com/kjkqjll)2月23日 回复 顶(1) 转发



kilin
2016年02月15日14:42:17
看了第二遍，留念。
感谢所有翻译人员！

2月15日 回复 顶 转发



Luke
回复 手机用户2577683113: 2 *2^3

2月9日 回复 顶 转发



Pas_mal (http://weibo.com/5576503322)
非常感谢翻译者.....

(http://weibo.com/5576503322)1月21日 回复 顶 转发



Pas_mal (http://weibo.com/5576503322)
新手，到了最后.

(http://weibo.com/5576503322)1月21日 回复 顶 转发



唯一
回复 手机用户2577683113: 10向左移3位是10000 。二进制的10000就是16

1月20日 回复 顶 转发



笑醉三千
已看完，Mark一下，还要实践


1月10日 回复 顶 转发



Xu
组合赋值运算符??

1月6日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»



说点什么吧...

发布